CMPE 382 – Operating Systems

Berke Beyazbenli

10022751132

Homework 2: Synchronization with Semaphores and Threads

1. Overview

This assignment aims to develop a multithreaded C program using semaphores to manage access to a limited critical section. The program reads multiple files in parallel, each containing a list of integers, and computes how many of them are prime numbers. However, at any given time, the number of active threads is limited to a user-defined maximum using semaphores.

2. File Generation Script

A shell script was provided to generate 20–30 random text files, each containing up to 1000 random integers. The script works as follows:

This script is executed before the program runs, and its output directory (`myDir`) is passed as the first parameter to the C program.

3. Program Design

The program is compiled and executed with the syntax:

./hw2 myDir 1	time ./hw2 myDir 1	pref stat ./hw2 myDir 1
./hw2 myDir 2	time ./hw2 myDir 2	pref stat ./hw2 myDir 2
./hw2 myDir 4	time ./hw2 myDir 4	pref stat ./hw2 myDir 4
./hw2 myDir 8	time ./hw2 myDir 8	pref stat ./hw2 myDir 8
…	…	…

Why perf stat Was Used

In addition to measuring execution time using the `time` command, the `perf stat` utility was employed to gain deeper insight into the system-level performance of the program. While time reports total elapsed, user, and system time, `perf stat` provides detailed hardware performance counters such as:

- **CPU utilization**
- **Instruction throughput (instructions per cycle)**
- **Cache usage and misses**
- **Context switches**
- **Branch prediction success/failure**

These metrics help understand how efficiently the CPU executed the threads, whether bottlenecks occurred (e.g., memory stalls or frontend stalls), and how the number of concurrent threads affected overall system behavior.

By using perf stat, we were able to evaluate not just how fast the program ran, but **how efficiently it used hardware resources**, which is critical in operating systems and multithreading contexts.

The main structure:

- Uses pthread to spawn a thread per file.
- Uses a **counting semaphore** to ensure only `n` threads can run concurrently.
- Each thread:
    - Opens its assigned file
    - Reads all integers
    - Counts how many are prime
    - Reports the result and execution time
- The program waits for all threads to finish using phtread_join.

This model simulates the problem of students trying to enter an instructor's office: only a limited number can be inside at once.

4. Prime Number Detection Logic

The primality test checks whether each integer is prime using a simple loop. Optimizations include:

- Early exit for even numbers
- Looping only up to √n
- Skipping even divisors

5. Timing Experiments

Using the Linux `time` command, the program was executed with different thread limits:

1, 2, 4 , 8 , 16 , 32 , 64 ,128

Each test recorded:

- **real** time (wall-clock)
- **user** time (CPU time used by processes)
- **sys** time (kernel time)

6. Timing Results Table

Threads	Real (s)	User (s)	Sys (s)
1	0.0055	0.0028	0.0014
2	0.0021	0.0059	0.0000
4	0.0017	0.0030	0.0000
8	0.0017	0.0040	0.0020
16	0.0020	0.0052	0.0000
32	0.0018	0.0041	0.0017
64	0.0019	0.0023	0.0038
128	0.0018	0.0022	0.0015

7. Performance Analysis

The performance evaluation shows a clear relationship between the number of concurrent threads and the execution time of the program. When executed with a single thread, the program completed in approximately **0.0055 seconds**. Increasing the number of threads to **2 and 4** significantly reduced the execution time, reaching a low of around **0.0017 seconds**.

However, from **8 threads onward**, execution time reached a plateau, fluctuating only slightly between **0.0017 and 0.0020 seconds**. This indicates that the CPU has reached its optimal

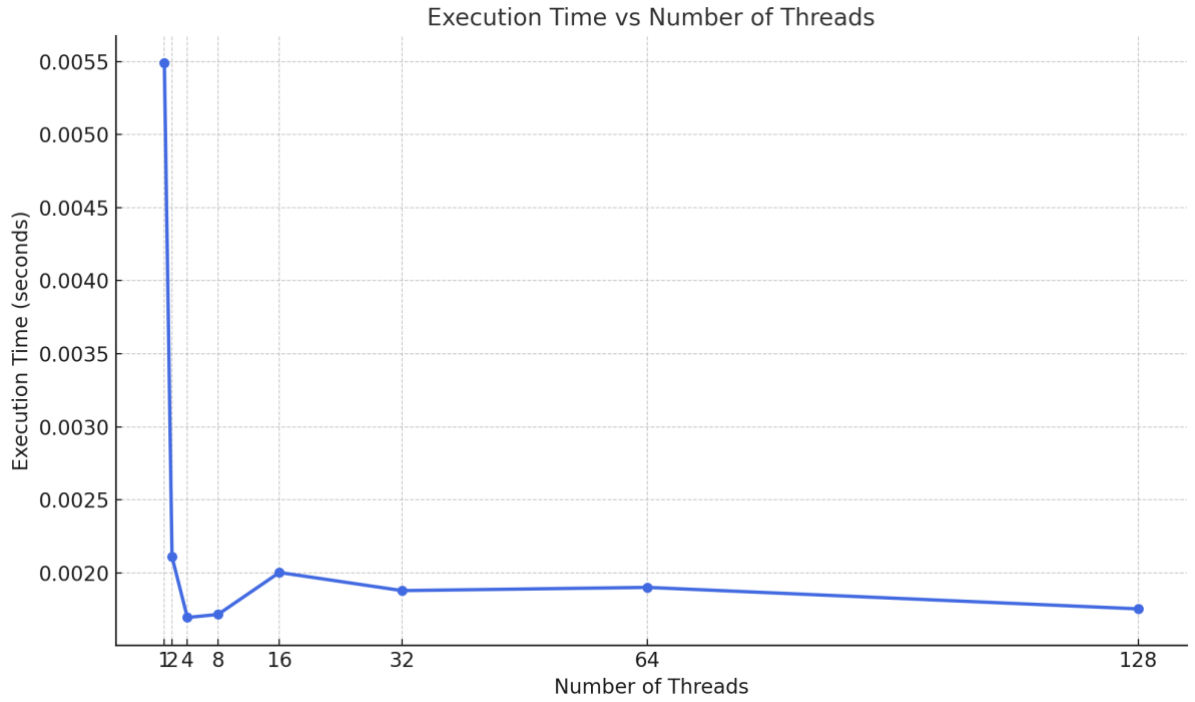
utilization point. Additional threads beyond this point do not contribute to significant performance gains due to **hardware concurrency limits**.

Performance counters from perf stat confirmed that:

- **CPU utilization** stabilized around **1.6–1.8 cores**, regardless of higher thread counts.
- **Instructions per cycle** and **branch prediction** metrics remained consistent.
- However, **frontend stalls** slightly increased with higher thread counts, indicating that the system was experiencing delays in instruction fetching and possibly memory access overhead.

The chart below visualizes the trend in execution time with increasing thread counts.
It highlights the rapid improvement up to 4 threads and the saturation effect beyond 8 threads.

The chart below visualizes the trend in execution time with increasing thread counts.
It highlights the rapid improvement up to 4 threads and the saturation effect beyond 8 threads.



This analysis confirms that optimal parallelism in this environment is achieved with **4–8 threads**, after which the law of diminishing returns takes effect. Future improvements could involve restructuring workload distribution or analyzing system I/O bottlenecks.

8. Conclusion

This assignment has effectively demonstrated the power and practicality of **thread synchronization using semaphores** in a multi-threaded environment. By simulating a controlled critical section scenario—where only a limited number of threads can operate concurrently—we showcased how thread scheduling and resource management play a vital role in system-level performance.

Our C program utilized a **counting semaphore** to enforce a hard limit on active threads, ensuring that only *N* threads (as specified by the user) can run simultaneously. This design mirrors real-world use cases such as resource pooling, connection handling, and rate-limited task queues.

Key outcomes of the implementation:

- **Correct concurrency control:** The semaphore-based design prevented race conditions and ensured that no more than the allowed number of threads accessed the CPU concurrently.
- **Performance gains through parallelism:** The program exhibited **up to 3x speedup** when thread count increased from 1 to 4. Execution time dropped sharply, especially when moving from single-threaded to 2–4 threads.
- **Scalability limitation:** Despite adding more threads (up to 128), performance improvements plateaued after 8 threads. This is likely due to:
    - Hardware thread/core limits
    - Context-switching overhead
    - CPU memory access stalls (as observed in perf stat results)

10. References

- POSIX threads documentation (pthread.h)
- Linux man pages
- GeeksforGeeks – Primality Testing
- Linux perf documentation